1. Queues and stacks can be thought of as specializations of lists that restrict which elements can be accessed.
1a. What are the restrictions for a queue?

The restrictions for a queue is that it can only insert from back of the queue and dequeue from front. First in First out. Cannot mutate the values inside of queue.

1b. What are the restrictions for a stack?

Restrictions for a stack is that it can only push from top and pop from top. First in Last out. Cannot mutate values inside of stack, also cannot access to first one to be placed until the last pop.

---

2. We have looked at lists backed by arrays and links in this class. Under what circumstances might we prefer to use a list backed by links rather than an array? (Your argument should include asymptotic complexity).

We prefer to use links rather than array when we know that we are going to add things many times. Since adding to the head or tail for linked list is O(1) and it's O(n) for the array because it has to copy all the contents to a temporary array that is a size bigger or smaller.

---

3. Give the asymptotic complexity for the following operations on an array backed list. Also provide a brief explanation for why the asymptotic complexity is correct.
3a. Appending a new value to the end of the list.

The upper bound is O(n), when it is not optimized it has to make a new list and copy each element from the old list to a new

list, but the size configuration of array is optimized, like make a significantly bigger array every time it reaches to a certain point, for the most of the case, the time complexity will be constant.

3b. Removing a value from the middle of the list.

O(n) things from middle to the tail should move its index by -1. For any case that i can think of for removing an element from middle, it would take a linear time because of the values after the removed value.

3c. Fetching a value by list index.
O(1) every single value has index allocated, so fetching with an index number is constant time for array.

---

4. Give the asymptotic complexity for the following operations on a doubly linked list. Also provide a brief explanation for why the asymptotic complexity is correct.

4a. Appending a new value to the end of the list.

O(1) only thing to do is to set Previous cell for the new cell and set Next cell for the last cell to a new cell and set last pointer to a new cell.

4b. Removing the value last fetched from the list.

O(1) fetching takes a O(n) time to do it, but removing takes only setting pointers of Prev and Next of the one before and one after to a new designated point, only takes four steps to do for any time.

4c. Fetching a value by list index.

O(n) none of the values have index allocated in them, so it has to go next until it hits the index number.

---

5. One of the operations we might like a data structure to support is an operation to check if the data structure already contains a particular value.

5a. Given an unsorted populated array list and a value, what is the time complexity to determine if the value is in the list? Please explain your answer.

O(n) since it has to traverse all the contents of the tree until finding a value that it's looking for.

5b. Is the time complexity different for a linked list? Please explain your answer.

O(n) same because linked list has to go over all of the contents in a list as well.

5c. Given a populated binary search tree, what is the time complexity to determine if the value is in the tree? Please give upper and lower bound with an explanation of your answer.
O(n) - If nothing can be found, it has to go over all the things if the tree was not sorted. - However, if the tree was sorted in some kind of order, the optimized time complexity would be O(log n).
Ω(1) - If the root was what we were looking for, there is only 1

5d. If the binary search tree is guaranteed to be complete, does the upper bound change? Please explain your answer.

Even if it is guaranteed to be complete, it has to be sorted to actually change the upper bound. If it is sorted, then it will change

the upper bound to O(log n) but, if it is not sorted, then it still remains O(n)

---

6. A dictionary uses arbitrary keys retrieve values from the data structure. We might implement a dictionary using a list, but would have O(n) time complexity for retrieval. Since we expect retrieval to occur more frequently than insertion, a list seems like a poor choice. Could we get better performance implementing a dictionary using a binary search tree? Explain your answer.

If the binary search tree is sorted, this will make it faster since sorted binary tree can eliminate half the options every time it moves to next branch. This will give a O(log n) time complexity for a search by a binary tree.